# Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah …

## 1 EINLEITUNG

Blah …

## 2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Blah …

## 3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

### 3.1 Monolithic Architecture

Der Begriff *Monolith*, abgeleitet aus dem Altgriechischen und mit der Bedeutung *einheitlicher Stein*, wird in der Softwarearchitektur verwendet, um ein Designmuster zu beschreiben, bei dem die gesamte Funktionalität einer Anwendung in einem einzigen, zusammenhängenden System integriert ist. In dieser Architektur übernimmt ein einzelner Prozess die Ausführung der gesamten Anwendung [Chen et al. 2017, 1].

Anwendungen, die diesem Architekturansatz folgen, bestehen aus eng miteinander verknüpften Komponenten, die wechselseitig voneinander abhängen. Diese Komponenten können weder unabhängig betrieben noch, in bestimmten Fällen, isoliert kompiliert werden [Salaheddin and Ahmed 2022, 485].


Abb. 1. Monolith Architektur

Diese Architektur bietet insbesondere für kleinere Anwendungen mehrere Vorteile, wie etwa eine vereinfachte Testbarkeit, Logging, Deployment und Debugging. Ein weiterer Vorteil besteht darin, dass keine separate Datenbanksynchronisation erforderlich ist, da sämtliche Daten in einer einzigen Datenbank gespeichert werden [Blinowski et al. 2022, 2].

Betrachten wir das E-Commerce-Beispiel. Dafür definieren wir drei Klassen:

- `OrderService`: Klasse, die die Bestellungen verwaltet
- `PaymentService`: Klasse, die die Zahlungen abwickelt
- `ShipmentService`: Klasse, die die Lieferungen initiiert

Die Kommunikation zwischen den Klassen erfolgt über Methodenaufrufe. In diesem Zusammenhang stellt die Klasse `OrderService` die zentrale Komponente dar, die die Methoden anderer Klassen nutzt, um den Bestellprozess zu steuern. Ein wesentlicher Vorteil dieses Ansatzes liegt in der einfachen Kommunikation zwischen den Komponenten. Durch den direkten Einsatz von Methodenaufrufen wird die Komplexität verringert, die typischerweise mit der Interaktion zwischen verschiedenen verteilte Systemen verbunden ist. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [1] zu finden.

Mit dem Wachstum und der zunehmenden Komplexität der Code-Basis treten jedoch einige Nachteile auf. Änderungen an einer Komponente können unerwartete kaskadierende Fehler auslösen, was die Weiterentwicklung in kleinen, autonomen Teams erschwert und verlangsamt.

Ein weiterer Nachteil ergibt sich aus der engen Kopplung zwischen den Komponenten, die durch direkte Methodenaufrufe entsteht. Änderungen an der Implementierung oder der Signatur einer Methode können in der Folge die gesamte Anwendung beeinflussen.

Zudem wird die Wiederverwendung von Funktionalitäten erschwert, da die Komponenten stark miteinander verknüpft sind. Dies führt zu einer Duplizierung von Code, was die Kosten für Änderungen erhöht, da diese an mehreren Stellen vorgenommen werden müssen.

Die daraus resultierende Komplexität und die erschwerte Wartung führen zu längeren Iterationen. Da das Deployment nur als Ganzes erfolgen kann, und die Iterationen sich verlängern, kommt es zu seltenen Auslieferungen neuer Features.

Ein weiteres Problem liegt in der erschwerten horizontalen Skalierung, da die Anwendung nur als Ganzes skaliert werden kann. Dies stellt eine Herausforderung dar, da bei wachsenden Anforderungen die gesamte Anwendung skaliert werden muss, anstatt einzelne Komponenten unabhängig voneinander zu skalieren.

Insgesamt ist die monolithische Architektur eine geeignete Lösung für kleinere Anwendungen, jedoch weniger geeignet für größere Systeme, da sie die Agilität und Flexibilität erheblich einschränken kann.

### 3.2 Modular Monolith

Das Hauptproblem der zuvor beschriebenen Architektur liegt in der engen Kopplung der Komponenten, was zu einer erhöhten Komplexität und begrenzten Flexibilität führt. Die Modular Monolithic Architecture stellt eine Weiterentwicklung der Monolithic Architecture dar, indem sie die Vorteile des monolithischen Ansatzes bewahrt und gleichzeitig die Nachteile der engen Kopplung verringert.

In diesem Architekturmodell wird die Code-Basis in mehrere Module unterteilt, die jeweils eine spezifische Teilfunktionalität der Anwendung implementieren.[Su and Li 2024, 11].

Die Modularisierung trägt zudem zur Reduzierung der Komplexität bei und ermöglicht eine bessere Organisation der Code-Basis, was sich positiv auf die Wartbarkeit der Anwendung auswirkt [Barde 2023, 23 - 24].

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.


[1] https://github.com/Beleg-6-EAP/demo-monolith-ecommerce


Abb. 2. Modular Monolith Architektur

Betrachten wir erneut das E-Commerce-Beispiel. Diesmal wird die Anwendung in drei Hauptmodule aufgeteilt (Siehe Abb. 2):

- `order`: Modul, das für die Verwaltung der Bestellungen verantwortlich ist
- `payment`: Modul, das für die Abwicklung der Zahlungen verantwortlich ist
- `shipment`: Modul, das für die Initiierung der Lieferungen verantwortlich ist

Wie die Abbildung 2 zeigt, jedes Service ist in einem eigenen Modul gekapselt. Eine beispielhafte Implementierung der PaymentService innerhalb des Moduls `payment` ist im Anhang **??** zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [2] zu finden.

Durch die Modularisierung sind die Komponenten weniger eng miteinander gekoppelt, was eine verbesserte Zusammenarbeit in kleinen, autonomen Teams fördert, im Vergleich zu traditionellen monolithischen Ansätzen.

Ein Nachteil bleibt jedoch bestehen: Die Anwendung muss weiterhin als Ganzes deployed werden, was die Iterationen verlangsamt. Zudem bleibt die horizontale Skalierung weiterhin erschwert, und die Funktionalitäten können nicht wiederverwendet werden, da sie immer noch Teil einer einzigen Anwendung sind und eng miteinander gekoppelt bleiben.

## 3.3 Layered

Das Problem der redundanten Schnittstellenlogik in Microservices lässt sich durch die Anwendung der Layered Architecture effektiv lösen. Dieses Architekturmuster basiert auf der Grundidee, eine Anwendung in verschiedene Schichten (englisch *layers*) zu gliedern. Dabei bleibt sowohl die Anzahl der Schichten als auch deren spezifische Aufgaben flexibel und wird nicht durch das Muster vorgeschrieben [Tu 2023, 34].

Diese Flexibilität erlaubt es Unternehmen, die Schichtstruktur individuell an ihre spezifischen Anforderungen anzupassen. Um jedoch Vorteile wie eine verbesserte Wartbarkeit, Skalierbarkeit und Wiederverwendbarkeit zu realisieren, ist die Einhaltung zentraler Prinzipien der Layered Architecture essenziell [Tu 2023, 34].

Ein grundlegendes Prinzip ist die Trennung der Zuständigkeiten (englisch *Separation of Concerns*). Hierbei werden Komponenten mit unterschiedlichen Aufgaben in separate Schichten aufgeteilt, sodass jede Schicht ausschließlich für eine klar definierte und abgeschlossene Funktionalität verantwortlich ist [Tu 2023, 34].

Ein weiteres wesentliches Prinzip ist die Isolation der Schichten (englisch *Layers of Isolation*), welches gewährleistet, dass Änderungen innerhalb einer Schicht lediglich deren eigene Komponenten betreffen und keine Auswirkungen auf andere Schichten haben [Richards 2015, 3–4].

[2] https://github.com/Beleg-6-EAP/demo-modulith-ecommerce

Die Layered Architecture bietet insbesondere für Unternehmen die Möglichkeit, häufig genutzte Funktionalitäten, wie etwa Authentifizierung oder Logging, in dedizierten Schichten zu kapseln und diese flexibel in verschiedenen Diensten wiederzuverwenden. Dabei sorgt eine klar definierte Schnittstelle jeder Schicht sowohl für eine reibungslose Kommunikation als auch für die Abstraktion der internen Implementierung.

Zusätzlich lässt sich die Layered Architecture nahtlos mit modernen agilen Architekturmustern wie Microservices kombinieren, wodurch die Agilität eines Unternehmens weiter gesteigert werden kann.

Ein praxisnahes Beispiel hierfür ist das E-Commerce-Beispiel mit Microservices. In der ursprünglichen Implementierung enthielt jeder Service ein eigenes Modul für die Authentifizierung. Diese redundante Struktur führte jedoch zu erheblichen Herausforderungen hinsichtlich Wartbarkeit und Kosten: Änderungen an der Authentifizierungslogik mussten in jedem einzelnen Service separat durchgeführt werden, was den Entwicklungsprozess verlangsamte und unnötig komplizierte.

Eine Lösung für dieses Problem besteht darin, die Authentifizierungslogik in eine zentrale Schicht auszulagern. Diese Schicht kann direkt mit dem ein API-Gateway kommunizieren und somit eine einheitliche Authentifizierung gewährleisten (siehe Abb. 3).

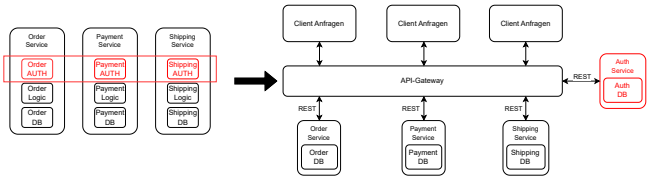


Abb. 3. Layered Microservice Architecture: Beispiel E-Commerce

Mit der Einführung einer zentralen Authentifizierungsschicht kann die Authentifizierungslogik an einer einzigen Stelle gebündelt und bei Bedarf angepasst werden, ohne Änderungen in den einzelnen Services vornehmen zu müssen. Dies bietet insbesondere in Unternehmen mit einer Vielzahl von Services signifikante Zeit- und Kosteneinsparungen.

Ein weiterer Vorteil besteht darin, dass die Authentifizierung für sämtliche Dienste zentral über ein API-Gateway durchgeführt wird, wodurch eine wiederholte Authentifizierung auf Service-Ebene entfällt. Diese Entlastung erlaubt es den einzelnen Services, sich ausschließlich auf ihre Kernfunktionalität zu konzentrieren, während die Authentifizierung ausgelagert ist.

Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [3] zu finden.

Die klare Trennung der Verantwortlichkeiten fördert die Zusammenarbeit in kleinen, autonomen Teams, da jedes Team unabhängig an einer Schicht arbeiten kann. Zudem ermöglicht das Prinzip der Isolation der Schichten eine höhere Flexibilität gegenüber wechselnden Anforderungen. Durch die Wiederverwendbarkeit in der

[3] https://github.com/Beleg-6-EAP/demo-microservice-ecommerce

Layered Architecture wird Mehraufwand vermieden und duplizierter Code reduziert, was zu kürzeren Iterationen und häufigeren Auslieferungen führt.

Das Beispiel verdeutlicht zudem, dass die Layered Architecture sich hervorragend mit agilen Architekturen wie Microservices kombinieren lässt. Diese Synergie trägt entscheidend dazu bei, die Agilität des Systems nachhaltig zu steigern.

## 4 MODERNE ENTERPRISE-ARCHITEKTUREN

### 4.1 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier „*An introduction to Software Architecture*“ auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Reagiert auf publizierte Events
- Vermittler (englisch *Mediator*): Liegt zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als ein Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 4 stellt diesen Vertrag dar.

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht agilen Entwickler-Teams durch klar abgegrenzte Features einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung maximale Skalierung. Daher sind Event-Driven Architekturen besonders für datenintensive

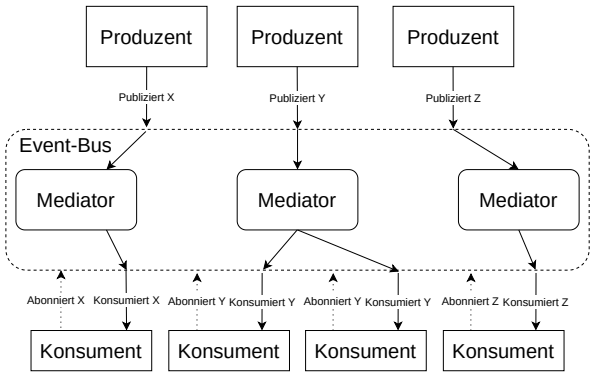


Abb. 4. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Die Agilität der Architektur kann weiter erhöht werden, indem der event-basierte Aspekt mit weiteren agilen Strukturen wie Microservices oder cloud-nativen Serverless-Functions kombiniert wird. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

## 5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

## 6 DISKUSSION

## 7 ZUSAMMENFASSUNG UND AUSBLICK

## LITERATUR

Kalpesh Barde. 2023. Modular Monoliths: Revolutionizing Software Architecture for Efficient Payment Systems in Fintech. (2023). https://doi.org/10.1145/3643657.3643911

Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. 2022. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* 10 (2022), 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 466–475. https://doi.org/10.1109/APSEC.2017.53

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015. *Software architecture patterns.* Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA ….

Nada Salaheddin and Nuredin Ahmed. 2022. MICROSERVICES VS. MONOLITHIC ARCHITECTURE [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]. *MINAR International Journal of Applied Sciences and Technology* 4 (10 2022), 484–490. https://doi.org/10.47832/2717-8234.12.47

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

Ruoyu Su and Xiaozhou Li. 2024. Modular Monolith: Is This the Trend in Software Architecture?. In *Proceedings of the 1st International Workshop on New Trends in Software Architecture* (Lisbon, Portugal) *(SATrends '24)*. Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/3643657.3643911

Zhenan Tu. 2023. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management* 11 (11 2023), 34–38. https://doi.org/ojs/index.php/jceim/article/view/14398

# A ANHANG 1

## A.1 Übungsaufgaben

Blah …

# B ANHANG 2

Blah …